LSAI Project Feature: FP8 Training

Ahmad Khan
ETH Zurich
ahkhan@ethz.ch

May 2025

1 Feature Description

In this feature, we enable FP8 training for our llama-like [Tou+23] model. FP8 precision reduces memory usage and bandwidth requirements by representing values with only 8 bits, allowing for more efficient computation on modern GPUs. By converting key components of the model to use FP8 arithmetic, we aim to accelerate training and reduce hardware resource consumption, while maintaining numerical stability and model performance through appropriate scaling and precision management. This is particularly useful in the grace hopper GPUs we use, since they have a specialized float8 computation capability.

2 Experiment Design

To evaluate the impact of FlashAttention, we conduct training experiments on the Swiss Alps supercomputer cluster [Swi24]. The model is trained with a sequence of length 4096 using a batch size of 1, a learning rate of 5e-5, and a warm-up period of 100 steps, for a total of 1000 training steps. We record metrics including throughput, TFLOPs, and loss at regular intervals.

To compare the performance of float8 comutation, we conduct 4 experiments:

1. Baseline: unchanged starter code
2. Baseline (compiled): unchanged starter code with the compile flag set to true
3. Rowwise FP8 (compiled): FP8 training in rowwise mode, with compile flag set to true
4. Tensorwise FP8 (compiled): FP8 training in tensorwise mode, with compile flag set to true

We make use of the `torchao` implementation of FP8 training[1]. `torchao` provides two different scaling strategies to preserve numerical accuracy while benefiting

[1] https://github.com/pytorch/ao/tree/main/torchao/float8

from 8-bit precision. In tensorwise scaling, a single scale and zero point are applied across the entire tensor, offering faster computation but less adaptability to local value ranges. In contrast, rowwise scaling applies separate scaling factors to each row (or sub-tensor), allowing finer-grained control and improved accuracy for tensors with high dynamic range, at the cost of slightly more overhead. These modes help balance performance and precision when training with FP8.
We use compiled mode[2] for both the scaling strategies. The reason for this is that one of the key features of `torchao` is that it can run in compiled mode. Additionally, when we run without compiled mode, the fp8 training is highly unoptimized and runs too slowly. The tasks time out past the 15min timelimit set. Therefore, we use the compiled mode, and add the baseline compiled as well for a fair comparison.

All the code to reproduce our results can be found in our codebase[3].

3 Results

In this section, we compare the 4 experiments across different dimensions.

3.1 Training Loss

Here we look at the training loss across the different experiments. The results are shown in fig. 1. We see that across all the training strategies, training loss is almost identical. This is a little surprising, since rowwise strategy is supposed to provide greater accuracy that tensorwise strategy, however, it is promising since it shows us that FP8 training works well in practice.

Figure 1: Training loss for different attention backends

[2]https://docs.pytorch.org/docs/stable/generated/torch.compile.html
[3]https://github.com/mak2508/lsai-proj

3.2 Tokens per second

Next we look at the tokens per second in fig. 2. Tokens per second measures the throughput of the model during training, indicating how many input tokens are processed per second—a higher value reflects faster training performance. This is relevant since it is can help us validate if the FP8 training is actually resulting in speed-ups. We see that tensorwise fp8 training gives us the fastest training, following by rowwise and then the 2 baseline results. This result follows expectation.

Figure 2: Tokens per second for different attention backends

3.3 Model FLOPs Utilization

Model FLOPs Utilization (MFU) measures the percentage of a GPU's theoretical peak floating-point operations per second (FLOPs) that are actually used by the model during training. It is a key efficiency metric that reflects how well the model implementation leverages hardware capabilities, with higher MFU indicating better performance utilization. This is presented in fig. 3. Here, we see again that tensorwise FP8 performs the best, followed by rowwise FP8 and then the baselines. This is expected since the grace hopper GPUs we use have specialized FP8 cores.

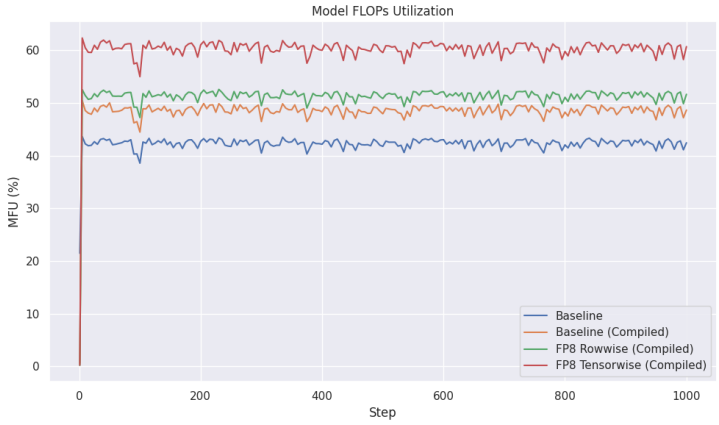

Figure 3: Model Flop Utilization (MFU) over training

3.4 Training Time

Finally, we look at the training time for each of the backends. We see in fig. 4 again, inline with the previous two results, that tensorwise fp8 and rowwise fp8 performs the best.

However, a caveat for this graph is that it seems the fp8 training takes a long time to warmup, upto 2mins in these experiments. I removed the time taken upto step 1 from these charts. If those times were included, then fp8 would no longer provide a benefit. However, this is a very short training run, and over a longer training scheme, this adjusted graph better tells us whether we achieve faster training or not.

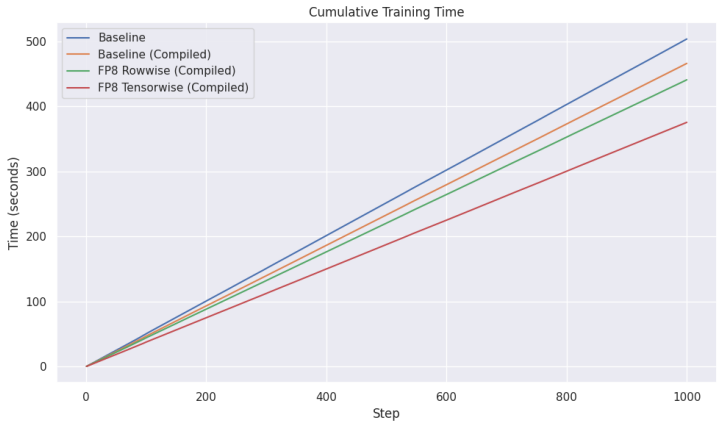

Figure 4: Training time across backends